

# 实践指导

吴清晏 (61522314)  
东南大学吴健雄学院

## 前言

感谢您使用《Guide to Labworks by WQY》!

本文由吴健雄学院 22 级吴清晏编写，希望能帮助同学们顺利完成操作系统实验的准备工作，加深对操作系统的理解。本文包含以下几个部分：

首先，课前准备部分，为使用 Windows 的同学提供了 Linux 虚拟机的安装教程，并根据自己在配置环境时遇到的问题和解决方法，希望能帮助同学们少走一些弯路。此外，还包括实验文件的下载以及 Xv6 的下载。后续可能还会添加其他内容。

## 目录

<b>1</b>	<b>课前准备教程</b>	<b>4</b>
1.1	虚拟机配置 . . . . .	4
1.2	其他配置 . . . . .	7
1.3	下载实验文件 . . . . .	9
1.4	下载 Xv6 . . . . .	10
<b>2</b>	<b>Guide to Lab works</b>	<b>11</b>
2.1	总体介绍——以 Reverse 为例 . . . . .	11
2.2	Kernel Hacking 介绍 . . . . .	15

# 1 课前准备教程

在开始前，需要明确你的需求：是仅仅想完成操作系统课程实验，还是想完全体验 Linux 的使用；是想仅使用命令行窗口，还是希望有一个完整的图形界面...无论如何，你都会需要的有：GNU 编译环境安装，实验文件下载，Xv6 配置。

此外，你可以选择在 `wsl` 和 `VMware` 中选择一个虚拟机平台（本教程仅针对 Windows 系统，因为 Mac 是 Unix 内核，不需要额外配置）

在这之后，你可以从 `VS code` 和其他文本编辑器（如 `vim`, `nano`, `Emacs` 等）中选择一种，个人推荐 `VS code`。

## 1.1 虚拟机配置

### 1.1.1 VMware 配置

如果你的需求仅仅是操作系统实验，那么 `wsl` 就足够了，占用空间更小，启动更方便，文件传输更便捷，但是！`wsl` 对 GUI(图形界面) 的支持极差，如果你需要在 Linux 环境下运行带有图形界面的软件（如使用 Open 3d 库的 Python 脚本），那么，你需要一个完整的 Linux 系统，包括图形界面。

网上有大量 VMware 的安装教程，这里推荐这一篇，安装完成后，你需要自己下载 Linux 安装镜像 (.iso 格式)，推荐通过东南大学最新搭建的镜像站下载，推荐 Ubuntu22.04 版本。

### 1.1.2 WSL + Ubuntu

注意！只有 Win10/11 才能使用 `wsl`！

1. 搜索启动或关闭 Windows 功能，打开适用于 Linux 的 Windows 子系统 和虚拟机平台，并根据提示重启。
2. 打开 Terminal，输入 `wsl --update`。

Tips:

有时会显示没有安装的分发版，首先按照提示输入 `wsl.exe --list --online` 查看可以安装的系统，输入 `wsl.exe --install < 系统名 >` 完成安装，推荐选择 Ubuntu

如果你的电脑近期重新安装过家庭版系统，有可能出现注册表缺失报错，解决方法为安装Win11 专业版，安装完成后在启动或关闭 Windows 功能页面勾选 Hyper-V，并重新打开两个功能。

3. 在安装过程中，提示输入用户名 (不能有大写) 和密码。

在 Linux 系统中，密码默认隐藏，记住自己输了几位！

如果未完成用户设置就关闭了 Ubuntu，将默认登录为 root 用户。

通过 `passwd` 命令设置 root 密码后，可通过 `adduser <username>` 添加用户，之后需要通过 `adduser <username> sudo` 命令给新用户添加管理员权限。

如果希望默认登录为普通用户 (而不是 root 用户)，可通过 `ubuntu config --default-user <username>` 设置默认登录用户，注意这一行命令不是在 Ubuntu，而是在 Windows 的 Terminal 或 Command 终端运行的。

(以上指令请替换 `<username>` 为用户名)

4. 在 Ubuntu 中，大部分命令都需要管理员权限 (sudo)，但现在还没有设置管理员 (root 用户) 密码，可通过 `sudo passwd` 设置，推荐采用与当前用户一样的密码，防止混淆。

### 1.1.3 GNU 编译环境安装

1. 进入 Ubuntu，输入 `sudo apt-get update` 更新，根据提示输入管理员密码。

2. 输入 `sudo apt-get install build-essential gdb`

build-essential 包括了 gcc,g++ 和 make, 其中 gcc 和 g++ 分别为 c 语言和 c++ 的编译器, make 可以编译带有 makefile 文件的开源软件代码。

GDB 的全称是 GNU Debugger，之后我们使用的 VS code 提供的断点调试等功能就是基于 GDB 的。

### 1.1.4 文本编辑器安装

Ubuntu 系统一般已经默认安装了 vim 和 nano，CentOS 系统一般只内置了 vi，但通过自带的包管理器可以很方便的安装。vim、emacs 和 nano 都是基于命令行的文本编辑器，而 Gnome 和 VS code 都拥有图形界面，可以使用鼠标辅助编辑，也可以粘贴多行文本。

vim 等命令行教程很多，重点是记住快捷键的使用，大部分 Linux 发行版已内置，无需额外安装。

Gnome 是 GNOME 桌面的默认文本编辑器，考虑到 wsl 对 GUI 的支持程度，个人感觉很鸡肋。如果有图形界面的文本编辑器，推荐直接使用 VS code。

在安装 wsl 版本的 VS code 前，需要先在 Windows 上安装 VS code，安装时可勾选添加到右键菜单

1. 在 wsl 中输入 `code` . 即可完成 VS code 安装，注意中间有空格。

(其实该命令主要用于在当前目录下启动 VS code，首次使用自动安装)

如果使用较老的系统版本 (如 CentOS7)，可能无法正常安装 VS code，因为最新版的 VS code 需要 Glibc 的版本大于等于 2.28。推荐 WSL 上不要折腾老系统，换个新点的 Linux。

2. 在 VS code 中选择文件-> 打开文件夹，输入 `~` 打开用户目录，新建 `test.c`，输入以下内容

(可根据提示安装 C/C++ 插件)

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Hello World\n");
    return 0;
}
```

3. 点击右上角按钮运行，在编辑器选项中选择 gcc，若输出为 Hello World，则说明 GNU 的配置正常。

### 1.1.5 配置 git

1. Ubuntu 默认已安装 git，只需配置用户名和邮箱 (改为自己的)
2. `git config --global user.name "Your Name"`
3. `git config --global user.email "youremail@domain.com"`
4. 如果 Windows 上没有安装 Git，点击链接下载并安装
5. `git config --global credential.helper "/mnt/c/Program\ Files/Git/mingw64/bin/git-credential-manager.exe"`
6. 可以尝试在 VS Code 中使用源代码管理进行推送与拉取
7. 如果把 Projects 放在 Git 仓库中，可通过在文件夹中添加.gitignore 文件，输入 `test*` 忽略所有测试用代码

默认情况下，WSL 会把虚拟机安装到 C 盘，但 C 盘往往空间比较紧张，如果希望把虚拟机安装到指定位置，可进行如下操作：

1. Windows Terminal 输入 `wsl --shutdown`(`wsl -l -v` 查看虚拟机状态)
2. 确认关闭后，输入 `wsl --export Ubuntu D:\wsl2.tar`(以 `D:\wsl2.tar` 为例)
3. 导出完成后卸载原虚拟机 `wsl --unregister Ubuntu`(假设虚拟机名称为 Ubuntu)
4. `wsl --import Ubuntu D:\Ubuntu_WSL\ D:\wsl2.tar`(假设导入到 `D:\Ubuntu_WSL`)

(本部分参考了wsl 官方教程)

## 1.2 其他配置

这一部分**不推荐**执行，但如果确实有相关需求，或许能帮忙少走一些弯路。

### 1.2.1 在 Linux 中安装 Chrome 浏览器

1. 输入 `cd /tmp` 打开临时目录

2. `wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb`  
下载 Chrome 安装包
3. `sudo apt install --fix-missing ./google-chrome-stable_current_amd64.deb`  
安装
4. `sudo apt-get install ttf-wqy-zenhei` 安装中文字体
5. 输入 `google-chrome` 启动浏览器

### 1.2.2 安装 VS code 插件 (快捷打开网页)

输入 `code` . 打开 VS Code, 在拓展程序部分, 搜索并安装插件 `Open Browser Preview`

Tips:

Chrome 自带了网页翻译功能, 若未能检测出英文网页, 按 `F12` 进入开发者工具, 选中 `<html>` 右键添加属性 `lang="en"` 将网站标注为英文)

### 1.2.3 配置中文输入法 (强烈不推荐)

如果你希望在 Linux 的浏览器上像 Windows 系统上一样输入中文, 可以尝试以下配置。

该过程比较危险, 推荐在尝试前先导出备份

1. 配置中文语言包

```
sudo apt install language-pack-zh-hans
```

2. 编辑 `/etc/locale.gen`, 去掉 `en_US.UTF-8 UTF-8` 及 `zh_CN.UTF-8 UTF-8` 前的注释符号

```
vim /etc/locale.gen
```

(按 `i` 编辑, `ESC+:wq` 保存并退出)

```
sudo locale-gen --purge
```

3. 安装输入法

```
sudo apt install fcitx fonts-noto-cjk fonts-noto-color-emoji dbus-x11
```



#### 4. 安装输入模式

```
sudo apt install <Package>
```

其中 package 从 `fcitx-libpinyin`, `fcitx-sunpinyin`, `fcitx-googlepinyin` 中挑选一个

#### 5. 切换到 root 用户，并创建 bus 连接

```
su root
dbus-uuidgen > /var/lib/dbus/machine-id
```

#### 6. 创建新文件 `vim /etc/profile.d/fcitx.sh`，输入

```
#!/bin/bash

export QT_IM_MODULE=fcitx
export GTK_IM_MODULE=fcitx
export XMODIFIERS=@im=fcitx
export DefaultIMModule=fcitx

#optional
fcitx-autostart &>/dev/null
```

#### 7. 在 Windows 终端中通过 `wsl --shutdown+wsl` 重启虚拟机

#### 8. 输入 `fcitx-config-gtk3`，不出意外的话，界面上出现之前安装的输入法。可通过 Global Config 调整切换输入法的快捷键。(如果失败，只能从导出的备份重新尝试)

#### 9. 打开浏览器，验证输入法功能是否正常

```
google-chrome
```

### 1.3 下载实验文件

#### 1. 点击链接下载并解压 `labworks`，重命名为 `projects`

```
cd ~
code .
```

#### 1. 将文件夹从 Windows 文件资源管理器拖到 VS code 左侧的文件窗格中，右键选择在资源管理器中打开。接下来就可以像使用 Windows 一样打开

`reverse` 文件夹，点击 `README.html` 查看实验说明。

如果之前配置了 Linux 浏览器和 VS code 插件，可选中文件并右键，选择

`Preview In Default Browser` 在 Chrome 中打开网页。

另一种方案：直接克隆本仓库(不推荐，因为仓库设置了 `gitignore`，文件不全)

## 1.4 下载 Xv6

Xv6 在后续的实验中将被使用，但它并不包含在刚刚的实验文件中。

1. 在 `Xv6-Syscall` 文件夹中打开终端

(`wsl` 使用 VS code 的集成终端会更方便，常用快捷键与 Windows 相同)

```
git clone https://github.com/mit-pdos/xv6-public.git
```

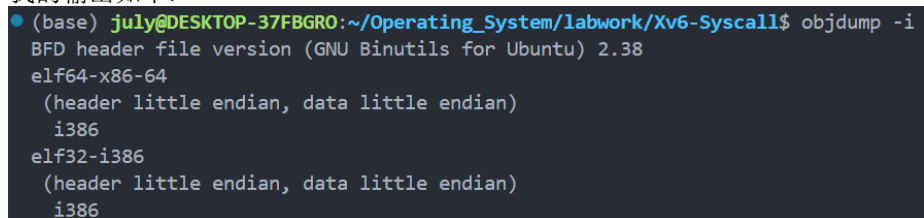
有同学反馈 `git` 仓库无法连接，建议使用校园网或流量热点，一般都是可以直接连接的。

如果要将克隆的仓库上传到自己的仓库中，需要删除该仓库的 `.git` 隐藏文件夹。

2. 测试编译工具

```
objdump -i
```

我的输出如下：



```
(base) july@DESKTOP-37FBGR0:~/Operating_System/labwork/Xv6-Syscall$ objdump -i
BFD header file version (GNU Binutils for Ubuntu) 2.38
elf64-x86-64
(header little endian, data little endian)
i386
elf32-i386
(header little endian, data little endian)
i386
```

如果第二行和我一样是 `elf32-i386` 就没问题了。

如果正常完成 GNU 配置，`gcc` 版本一定不会有问题的。

3. 编译 xv6

打开刚刚克隆的文件夹，例如 `xv6-public`，再运行 `make` 编译

```
cd xv6-public
```

```
make
```

`make` 包含在之前安装的 `build-essential` 包中，当目录下有 `makefile` 文件时会自动根据文件指示逐行编译（若指明的 `.c` 文件已经编译为 `.o` 则跳过，不会重复编译）。在后续我们添加系统调用的过程中，如果添加了新的 `.c` 文件，需要修改 `makefile`，加入新文件名，这样该文件可以在后续的 `qemu` 虚拟机窗口中作为可执行的命令被调用。

#### 4. 安装 `qemu` 虚拟机

```
sudo apt-get install qemu-system
```

(原先的命令已过时，官网已经更新了安装方式)

#### 5. 用虚拟机启动 `Xv6`

```
make qemu
```

## 2 Guide to Lab works

### 2.1 总体介绍——以 `Reverse` 为例

在 VS code 中打开 `~` 目录，在 `./projects/Reverse` 下新建 `reverse.c` 文件。

#### 2.1.1 需求分析

##### 1. 支持 3 种输入形式：

- `./reverse`
- `./reverse input.txt`
- `./reverse input.txt output.txt`

##### 2. 对于输入的数据（命令行/文件），不能假设句子长度和句子个数。

##### 3. 处理以下 4 种错误：

- 输入参数过多：usage: reverse <input> <output>

- 文件无法打开: `reverse: cannot open file '<filename>'` (其中 `<filename>` 为打不开的文件名)
- 输入相同文件: `reverse: input and output file must differ` (不能仅通过文件名判断)
- 内存分配失败: `malloc failed`

无论是哪一种错误, 统一用 `fprintf(stderr, "<error message>\n");` 输出错误并 `exit(1);` 返回状态码 1。

其中, `stderr` 是一种特殊的输出流, 与之类似的输出流是 `stdout`, `stdout` 类似 `c++` 中 `cout`。

返回的状态码正常为 0, 调用 `exit` 函数会立即终止并返回指定状态码。

### 2.1.2 功能实现

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
```

#### 1. 处理错误“输入参数过多”

```
// 如果用户运行时 reverse 参数过多, 则打印 usage: reverse <input> <output> 并退出, 返回
if (argc > 3) {
    fprintf(stderr, "usage: reverse <input> <output>\n");
    exit(1);
}
```

#### 2. 处理错误“文件无法打开”

```
// 输入流, 文件或命令行输入 (Ctrl+D 结束输入)
FILE *input = stdin;

// 输出流, 文件或命令行输出
FILE *output = stdout;

// 如果提供输入文件, 打开输入文件
```

```

if (argc >= 2) {
    input = fopen(argv[1], "r");
if (input == NULL) {
    fprintf(stderr, "reverse: cannot open file '%s'\n", argv[1]);
    exit(1);
}
}

```

// 如果额外提供输出文件，尝试打开，并检查输入输出文件是否相同（用 `stat` 防止硬链接）

```

if (argc == 3) {
    output = fopen(argv[2], "w");
if (output == NULL) {
    fprintf(stderr, "reverse: cannot open file '%s'\n", argv[2]);
    exit(1);
}
}

```

3. 通过头文件 `<sys/stat.h>` 提供的 `stat` 函数处理错误“输入相同文件”

```

struct stat stat1, stat2;
stat(argv[1], &stat1);
stat(argv[2], &stat2);
if (stat1.st_ino == stat2.st_ino) {
    fprintf(stderr, "reverse: input and output file must differ\n");
    exit(1);
}

```

4. 分配初始内存，当容量不够时自动扩容，处理错误“内存分配失败”

```

// 记录行数
int num_lines = 0;

// 记录容量，初始为 10
int capacity = 10;

// 用于存储行的数组

```

```

char **lines = malloc(capacity * sizeof(char *));
if (lines == NULL) {
    fprintf(stderr, "malloc failed\n");
    exit(1);
}

size_t len = 0;
while (1) {
    if (num_lines == capacity) {
        capacity *= 2;
        lines = realloc(lines, capacity * sizeof(char *));
    }
    if (lines == NULL) {
        fprintf(stderr, "malloc failed\n");
        exit(1);
    }
    if (getline(&lines[num_lines], &len, input) == -1)
        break;
    num_lines++;
}

```

getline 函数在 len 设置为 0 时，会自动扩充输入缓冲区并更新 len 参数。

如果想通过终端测试零参数下的效果，可通过 Ctrl+D 终止输入流，此时 getline 函数会返回-1。

5. 将获取的所有句子逆序放入输出流，释放内存并关闭文件

```

for (int i = num_lines - 1; i >= 0; i--) {
    fprintf(output, "%s", lines[i]);
    free(lines[i]);
}

free(lines);

```

```
if (input != stdin)
    fclose(input);
if (output != stdout)
    fclose(output);
```

注意，与 C++ 不同，C 语言中用 malloc 分配的内存一定要主动调用 free 函数进行内存释放，文件需要主动关闭，这是比较好的代码习惯。

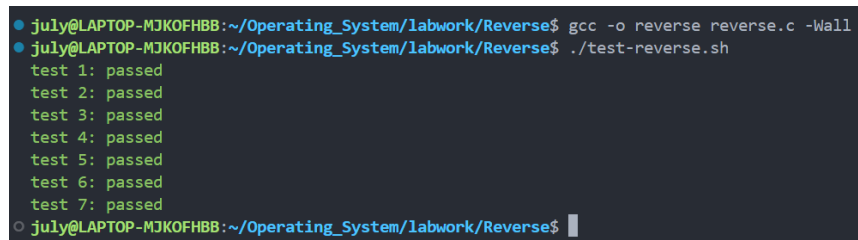
### 2.1.3 编译文件并测试功能

1. 选中文件，右键“在集成终端中打开”
2. 输入 `gcc -o reverse reverse.c -Wall` 进行编译
3. 输入 `sudo chmod 777 test-reverse.sh` 对当前测试脚本的权限进行修改

(你可能还需要输入 `sudo chmod 777 ../tester/*` 将其他测试脚本的权限设为最高)

4. 输入 `./test-reverse.sh` 进行测试。

如果一切顺利的话，你会看到以下结果



```
july@LAPTOP-MJKOFHBB:~/Operating_System/labwork/Reverse$ gcc -o reverse reverse.c -Wall
july@LAPTOP-MJKOFHBB:~/Operating_System/labwork/Reverse$ ./test-reverse.sh
test 1: passed
test 2: passed
test 3: passed
test 4: passed
test 5: passed
test 6: passed
test 7: passed
july@LAPTOP-MJKOFHBB:~/Operating_System/labwork/Reverse$
```

## 2.2 Kernel Hacking 介绍

在 Xv6 的文件夹，有 makefile 和很多 .c, .h, .S 文件，在编译后额外出现了 .d 和 .o 文件。其中，.c 是我们比较熟悉的 c 语言，.S 或 .s 都是汇编文件的后缀名，其中 .S 后缀的文件支持预处理命令 (如 # 开头的大写命令)

在进行系统调用的过程中，我们主要关心的文件有：

文件名	功能简介
<code>usys.S</code>	提供用户态与内核态转换的接口
<code>syscall.h</code>	定义系统调用号
<code>syscall.c</code>	转发用户发起的系统调用到内核
<code>sysfile.c</code>	实现系统调用
<code>sysproc.c</code>	另一种实现系统调用的选项
<code>user.h</code>	定义系统调用的参数传递方式

接下来，我们逐个分析这些文件，查看系统原有的系统调用是如何实现的。

### 2.2.1 `usys.S`

可以看到，这个文件包含 3 个部分。

```
#include "syscall.h"
#include "traps.h"
```

首先在头文件中导入了 `syscall.h` 和 `traps.h`，前者稍后会提到，包含每个系统调用对应的编号，也就是这里的 `$SYS_ ## name`，后者提供了 `$T_SYSCALL` 的定义，在这个系统中被定义为 64。

拓展信息:

#### 1. `$T_SYSCALL`

在 x86 架构的计算机中，当一个系统调用发生时，会触发一个陷阱（trap）。陷阱是一种特殊的中断，它会将 CPU 从用户态切换到内核态，然后跳转到预设的处理函数执行。这个处理函数的地址是在中断描述符表（IDT）中查找的，而 `$T_SYSCALL` 就是在 IDT 中的索引。

#### 2. IDT

IDT 在 `trap.c` 中被实现，可以看到这是一个大小为 256，类型为 `gatedesc` 的数组。`gatedesc` 在 `mmu.h` 中被定义，这是一个结构体，包括了 `off_15_0` 和 `off_31_16`（合起来表示中断处理程序在其所在段的偏移地址），`cs`（指定处理函数所在



段), `args`(参数数量) 和其他信息, 在这个实验中该部分并不重要, 故不详细展开。

接下来, 使用宏定义了接口的通用模板, `.globl name` 使系统调用的名称成为全局变量, 用户程序无须导入任何头文件就可以直接使用该名称对应的函数。当该名称被使用时, 自动把当前系统调用对应的编号放入`%eax`, 并发起中断 (注意此处的 `int` 是 `interrupt` 的简写)

```
#define SYSCALL(name) \  
    .globl name; \  
    name: \  
        movl $SYS_ ## name, %eax; \  
        int $T_SYSCALL; \  
        ret
```

拓展信息:

#### 1. `%eax`

`eax` 的全称是 `Extended Accumulator Register`, 是 16 位的 `ax` 的 32 位扩展。寄存器可近似理解为 c 语言中的变量, 但是汇编中 `eax`, `ebx`, `ecx`, `edx` 并不相同, 也不存在 `eex` 或 `efx`。实际上, `b` 指 `base`(基底), `c` 指 `counter`, `d` 指 `data`。

在 x86 架构的 Linux 中, 系统调用的编号是通过`%eax` 寄存器传递的, 这是一个约定。调用完成后的返回也是由`%eax` 负责的。

#### 2. `int`

大家可能会将其与 c/c++ 中的 `int` 发生混淆。在汇编语言中, `int` 指令的格式为 `int n`, 其中 `n` 为中断类型码。执行中断指令时, 首先记录当前状态 (通过把 `IP` 指令指针和 `FLAGS` 标志寄存器入栈), `IF=0` 和 `TF=0`, 并根据 `n` 在 `IDT` 中查找对应陷阱门, 包含了中断处理函数的段选择子和偏移地址。接下来, 将中断门或陷阱门的段选择子加载到代码段寄存器 (`CS`), 将偏移地址加载到指令指针 (`IP`)。这样, CPU 就跳转到了中断处理函数。之后的操作不展开介绍, 当执行完成时通过 `iret`

指令弹出 IP 和 FLAGS，恢复 CPU 状态并继续执行指令。

### 3. IF(Interrupt Flag)

IF 属于 FLAGS 标志寄存器，当 IF 为 1 时，允许响应可屏蔽中断；当 IF 为 0 时，禁止响应可屏蔽中断。

### 4. TF(Trap Flag)

当 TF 为 1 时，CPU 会在执行每条指令后生成一个调试异常，这通常用于单步调试；当 TF 为 0 时，CPU 不会生成调试异常。

gdb 的单步调试往往就是基于这个实现的。

最后一部分使用了上面定义的宏，为了实现我们的 `getreadcount(void)` 函数，我们只需要模仿其他系统调用：

```
SYSCALL(getreadcount)
```

## 2.2.2 syscall.h

刚刚已经提到，这个文件用于定义系统调用号。我们只需要在最后加上一行 `getreadcount(void)` 对应的调用号即可，以 22 为例：

```
#define SYS_getreadcount 22
```

## 2.2.3 syscall.c

刚刚提到，`int` 指令使用了 `trap.c` 提供的 `trap(struct trapframe *tf)` 方法，其参数结构体 `trapframe` 传递了陷阱出现时的各种信息，感兴趣的可以自己在 `x86.h` 中查看。

```
if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
        exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
        exit();
}
```

```

return;
}

```

之后，如果中断向量号 `trapno`(调用 `int` 指令自动把 `n` 放入该位置) 是系统调用陷阱，之后保存 `tf` 信息并执行 `syscall` 函数，所以下一步我们需要修改的就是 `syscall.c`。在 `syscall.c` 中，我们找到了 `syscall()` 函数的定义：

```

void syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        curproc->tf->eax = syscalls[num]();
    }
    else
    {
        cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}

```

还记得 `eax` 的内容吗？对于我们的 `getreadcount` 来说，是 22，在上面定义了名为 `syscalls` 的列表，目前只有 21 行，很明显我们需要把 `getreadcount` 添加到这里：

```

[SYS_getreadcount] sys_getreadcount,

```

这时，出现了未定义标识符 `"sys_getreadcount"` 报错，继续向上翻，我们还需要添加一行：

```

extern int sys_getreadcount(void);

```

这一行会告诉编译器，我们已经在其他文件中定义了该函数，编译器会自动去寻找，这样会防止未定义报错，但编译器依旧警告找不到 `"sys_getreadcount"`

的函数定义。看起来我们还需要在 `sysfile.c` 或 `sysproc.c` 中完成函数的定义。注意, 尽管在这两个文件中定义中结果上是等价的, 但在逻辑上, `sysfile.c` 负责与文件系统相关的系统调用, 如打开文件、读写文件、关闭文件等, 而 `sysproc.c` 负责与进程管理相关的系统调用, 例如创建进程、结束进程、等待进程等。让我们看一下测试用例 `test_1.c`:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[])
{
    int x1 = getreadcount();
    int x2 = getreadcount();
    char buf[100];
    (void)read(4, buf, 1);
    int x3 = getreadcount();
    int i;
    for (i = 0; i < 1000; i++)
    {
        (void)read(4, buf, 1);
    }
    int x4 = getreadcount();
    printf(1, "XV6_TEST_OUTPUT %d %d %d\n", x2 - x1, x3 - x2, x4 - x3);
    exit();
}
```

很明显, `getreadcount` 被调用后会记录 `read` 系统调用的次数, 个人认为它应该被归类为文件操作, 当然, 这并不重要, 你也可以选择在 `sysproc.c` 中完成函数的定义, 操作上没有区别。

#### 2.2.4 `sysfile.c`

在测试用例 1 的预期输出中, 我们看到 `0 1 1000`, 这说明我们需要用一个全局变量存储 `read` 指令的执行次数, 修改 `read` 指令, 记录指令的调用次数, 最

后实现 `getreadcount` 函数，使得调用 `getreadcount` 时获取该值。

相信大家能自己完成该部分，不要参考网上的教程，否则你会减少很多乐趣，例如老师提到的测试用例 2 的错误。

### 2.2.5 user.h

这个文件定义系统调用的参数传递方式，我们要实现的系统调用没有参数，所以我们只需要增加以下代码到系统调用部分。

```
int getreadcount(void);
```

这样能帮助 c 编译器检查系统调用传递的参数是否正确。

### 2.2.6 测试功能

在文件夹 `Xv6-Syscal` 中打开终端。

首先，手动把刚刚的文件夹 `xv6-public` 改名为 `src`，并修改测试文件权限。

(假设 `../test` 文件夹中的脚本均已设为最高权限)

```
sudo chmod 777 test-getreadcount.sh
./test-getreadcount.sh
```

正常情况下，因为在 `getreadcount` 实现部分没有对多线程进行针对性的处理，测试用例 2 可能会存在一种极端情况，两个进程同时读取了 `readcount` 的值并执行 `++` 操作，这样他们写回的值将会是原先加一，而不是加二，导致最后的结果不符，`test1` 通过，`test2` 不通过。但是，我这边即使反复尝试，甚至把次数增加到 100 倍，也没有出现同时访问导致的错误。

(本部分参考了 Xv6 官方的介绍和教学视频)